MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963 A

AD-A184 757

RADC-TR-87-103
In-House Report
July 1987

ERIC: AN OBJECT-ORIENTED SIMULATION LANGUAGE

DTIC
ELECTE
SEP 1 6 1987
S   D

Michael L. Hilton ,1Lt, USAF

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

87 9 15 109

87 9 15 109

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-87-103 has been reviewed and is approved for publication.
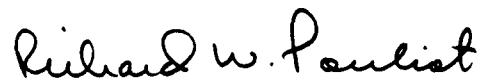
APPROVED: *(signature)*

SAMUEL A. DINITTO, JR.
Chief, C^2 Software Technology Division
Directorate of Command and Control

APPROVED: *(signature)*

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command and Control

FOR THE COMMANDER: *(signature)*

RICHARD W. POULIOT
Directorate of Plans & Programs

/ / / / /

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS N/A

2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	Approved for public release; distribution unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S) RADC-TR-87-103	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A

6a. NAME OF PERFORMING ORGANIZATION Rome Air Development Center	6b OFFICE SYMBOL (If applicable) COES	7a. NAME OF MONITORING ORGANIZATION N/A

6c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	7b. ADDRESS (City, State, and ZIP Code) N/A

8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center	8b OFFICE SYMBOL (If applicable) COES	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N/A

8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10 SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO. 61102F	PROJECT NO. 2304	TASK NO J4	WORK UNIT ACCESSION NO 15

11. TITLE (Include Security Classification)

ERIC: AN OBJECT-ORIENTED SIMULATION LANGUAGE

12. PERSONAL AUTHOR(S)
Michael L. Hilton, 1Lt, USAF

13a. TYPE OF REPORT In-House	13b. TIME COVERED FROM Jan 85 TO Oct 85	14. DATE OF REPORT (Year, Month, Day) June 1987	15. PAGE COUNT 60

16. SUPPLEMENTARY NOTATION

N/A

17.	COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computer Simulation Computers
09	02	15	Programming Language Object-Oriented Programming
09	05	30	Knowledge-Based Simulation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

ERIC is an object-oriented programming language designed to support intelligent simulations. This report describes the ERIC language. ERIC supports the construction of software "objects" that model objects in the physical world. These software objects can be combined to form simulations which are characterized by a high degree of flexibility and perspicuity.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Michael L. Hilton, 1Lt, USAF	22b TELEPHONE (Include Area Code) (315) 330-7794 22c OFFICE SYMBOL RADC (COES)

DD Form 1473, JUN 86   Previous editions are obsolete.   SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

PREFACE

ERIC is an object-oriented programming language designed for supporting the development of intelligent, discrete, event-driven simulations. ERIC was developed as part of an on-going research effort at the Rome Air Development Center to build a new generation of knowledge-based simulations that support Battle Management studies.

Object-oriented programming languages are designed to support the development and maintainence of large, complex software systems. These systems are composed of objects which have certain attributes and behaviors. Objects communicate with each other by message passing. The object-oriented paradigm is particularly useful for modelling and simulation because many real-world systems are composed of objects whose interactions can be represented by messages.

This report is a description of the ERIC programming language. It does not assume the reader is familiar with object-oriented programming or simulation; however, it does assume that the reader is familiar with Lisp.

i

TABLE OF CONTENTS

Chapter 1

Objects, Classes, and Inheritance

1.0 Objects

Objects are software entities that model real-world things. They combine the properties of both data and procedures: objects maintain a local state and are capable of performing computations using and/or modifying their local state.

The local state of an object consists of its attributes. Attributes describe characteristics of an object. For example, an object modelling an automobile might have color, engine size, and number of doors as attributes.

An object performs computation via behaviors. A behavior is a piece of procedural code associated with a given object or group of objects. Each behavior is identified by a particular message, which will invoke the behavior when the message is received by an object. For example, an automobile object might be sent a message telling it to start its engine. Message passing is how objects in ERIC interact with each other and the user. Message passing is one of the most important characteristics of ERIC and will be examined in more detail in Chapter 2.

There are two types of objects in ERIC: class objects and instance objects. A class is a description of one or more similar objects. In other languages, such as Pascal, classes correspond to types. An instance is an actual member of a class. For example, 3 is an instance of the class integer. A class can be a member, or subclass, of a more general class. Integers are a subclass of numbers. The ability to build more specific classes (such as integers) out of a more general class (numbers, in this case) is a powerful abstraction mechanism. ERIC provides this mechanism via class inheritance. A subclass can inherit attributes and behaviors from one or more superclasses.

[For the rest of this manual, use of the term "instance object" will be restricted to only those members of a class that are not themselves classes. In the numbers example given above, 3 will be considered an instance object, but integer will not. Also, the terms class and class object will be used interchangeably.]

1.1 Classes

Classes are used to model the organization of a system being simulated by an ERIC program. The system is decomposed into various real-world objects, and this decomposition is mapped onto software objects. More often than not, class objects are defined that are never actually instantiated. These class

objects exist purely for organizational reasons. This fact
reflects an important insight into the implementation of object-
oriented simulations: class objects should be used to organize
and manage the objects in a simulation; they should not be an
operational part of it. A careful examination of the system
being simulated is necessary in order to determine which classes
should be organizational in nature, and which classes should
actually be instantiated.

There is a predefined class in ERIC called **something**. All
other classes are built on top of (and hence, are subclasses of)
something. **Something** provides many basic behaviors for all
objects, such as how to print out attributes, make instances, or
define new behaviors. New classes are defined in ERIC by using
the *define-class* special form:

```
(define-class name
    (:parents superclass+)
  {(:documentation form)}
  {(:instance-attributes {variable |
                              (variable {init-value})}*)}
  {(:class-attributes   {variable |
                              (variable {init-value})}*)})
```

Example 1 shows several class definitions. This example will be
referred to several times in the following discussion.

The :parents list is the only required information for
defining a new class. It specifies the new class' inheritance
chain. This chain determines which attributes and behaviors will
be inherited from other classes. In Example 1, both fish and
mammal have **something** as the only superclass in their inheritance

3

chain. **Marine-mammal**, however, has three superclasses in its chain: **fish**, **mammal**, and **something**, in that order. Notice that **something** is included in the inheritance chain even though it was not explicitly included in **marine-mammal's** :parents list. The order of the superclasses in the inheritance chain is very important, because this order determines what attributes and behaviors an object inherits.

<u>Example 1</u>
<u>Sample Class Definitions</u>

```
(define-class fish
  (:parents something)
  (:class-attributes
   (environment 'water))
  (:instance-attributes
    genus
    species
    common-name
    length))

(define-class mammal
  (:parents something)
  (:class-attributes
   (environment 'land))
  (:instance-attributes
   genus
   species
   common-name
   (length 'long)))

(define-class marine-mammal
  (:parents fish mammal)
  (:instance-attributes
    (weight '2-tons)))
```

4

Three rules govern the ordering of the inheritance chain:

1. A class always precedes its own superclasses.
2. The local ordering of superclasses in every object's :parents list is preserved.
3. Duplicate classes are removed from the ordering.

In Example 1, the inheritance chain for fish and mammal are trivial; they are [fish, something] and [mammal, something], respectively. The inheritance chain for marine-mammal is a bit more complex. Following the above three rules, we will construct marine-mammal's inheritance chain. The chain begins with marine-mammal (Rule 1). Next, we add fish and its inheritance chain. So far, the (incomplete) chain is:

[marine-mammal, fish, something...

The next class in the parents list is mammal. Upon examining the inheritance chain of mammal, we discover that something is a superclass of mammal. According to Rule 1, we must place mammal before something. The chain is now:

[marine-mammal, fish, mammal, something...

Continuing, we add the rest of the inheritance chain for mammal, which consists of something, to the end of the partial chain. We now have:

[marine-mammal, fish, mammal, something, something]

According to Rule 3, the duplicate something is removed from the chain. The final result is the complete inheritance chain for marine-mammal:

[marine-mammal, fish, mammal, something]

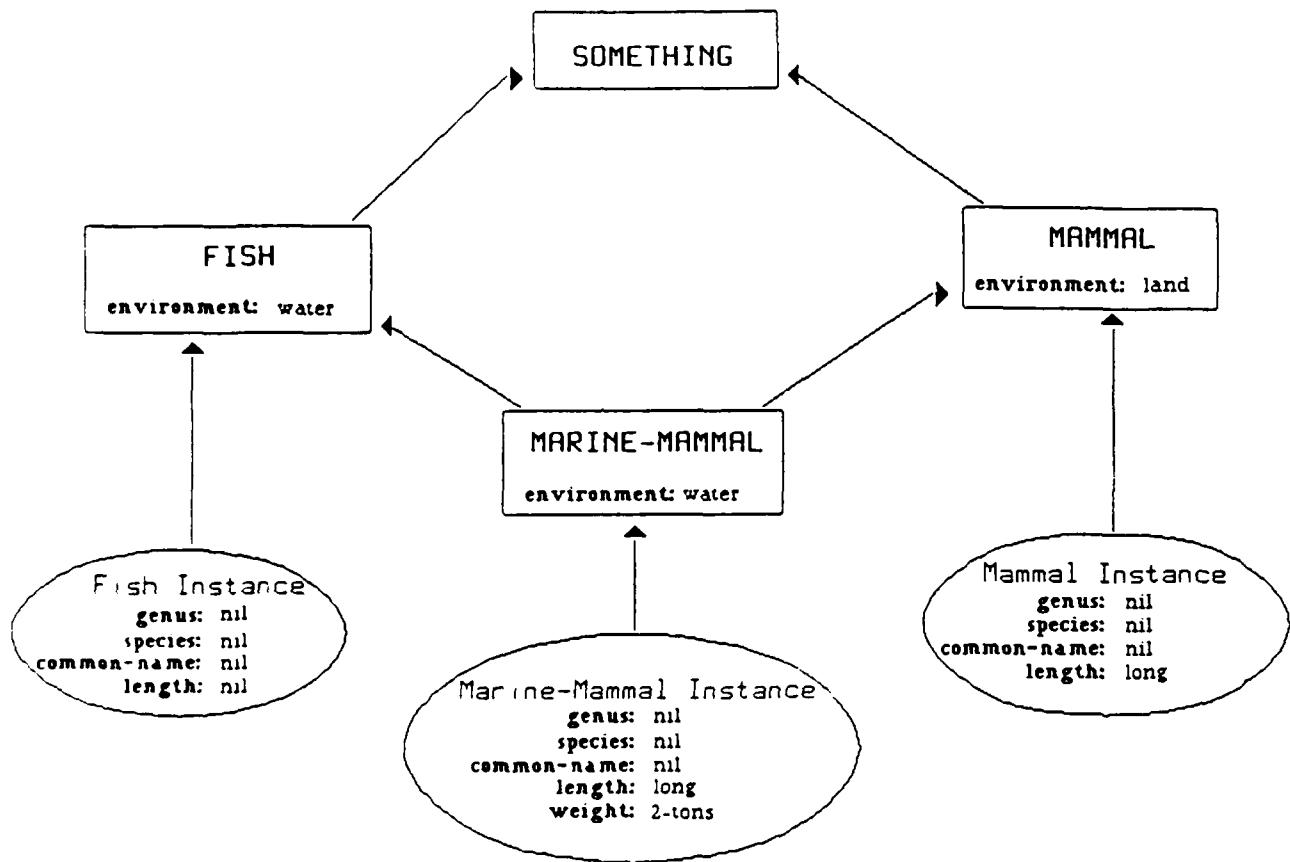Figure 1 shows the complete inheritance hierarchy for Example 1.

It is possible to write class definitions that cannot be ordered using the three rules. As an example, consider these class definitions:

```
(define-class A (:parents B C))
(define-class C (:parents B))
```

From the definition of object C, we find that C must precede B because a class always precedes its superclasses, but B must precede C in order to preserve the local ordering of superclasses in the parents list of A. When an inheritance chain cannot be computed according to the three rules, ERIC signals an error and prints out information detailing which classes are in conflict.

The :documentation list is an optional string that is used to document the class being defined.

The optional :instance-attributes list declares what new attributes an instance of the defined class will have in addition to those it gets via inheritance. The :instance-attributes list is composed of zero or more attribute names or name-value pairs, where value is a lisp form that is evaluated at the time of definition to a default initial value. If a default value is not provided, its default is NIL. Attributes are inherited by first taking the union of the instance variables of each class in the inheritance chain and then eliminating duplicates. When eliminating duplicates, precedence is given to the leftmost

6

This diagram shows the inheritance hierarchy of Example 1.
The class and instance variables inherited by each
object is shown inside the object's box.

<u>Figure 1</u>

attribute with a default value. (Leftmost being with respect to the inheritance chain.) In Example 1, an instance of **marine-mammal** will have the following attributes: *weight* (with a default value of 2-tons), *genus, species, common-name,* and *length* (with a default value of long).

Class attributes are declared by an optional :class-attributes list and are inherited in a way similar to instance attributes. Class attributes belong only to class objects; they are not inherited by instance objects. Class attributes can be used to store any type of information, but are most useful for storing information that is common to all members belonging to a class and for managing instance objects and subclasses.

The value returned by the define-class form is the actual class object being defined. This object is also bound to the symbol in the *name* position of the define-class form. This symbol is declared to be a global variable when the define-class form is evaluated, so you can always use this symbol as a "handle" to get ahold of the class object anywhere in your programs. You should be careful not to rebind this symbol to another value, or you may not be able to get ahold of the class object again.

Class objects identify themselves when printed as:

#<CLASS *x*>

where *x* is the name of the class, as specified in the class' define-class form. For example, the result of typing the form:

8

```
(eval marine-mammal)
```
at the top-level of a lisp system after Example 1 has been
evaluated will be the class object **marine-mammal**, which prints
itself as:
```
#<CLASS MARINE-MAMMAL>
```
There is also a predicate function, *class-object?*, of one
argument that will return true if an object is an ERIC class
object.

Every class object in ERIC has several predefined
attributes which may be inspected by users: *documentation*, the
string which may have been included in the :documentation list of
the class' define-class form; *parents*, the :parents list from
the define-class form; *offspring*, a list of all the immediate
subclasses and instances of this class; and *status* and *schedule*,
whose functions will be explained in Chapter 4. The value of
these attributes may be used in your programs, but you should
never try to change their values. These attributes are
established and maintained by ERIC; any modification by a user
could (and probably will) cause problems in the evaluation of
programs.

1.2 Instance Objects

Instances are the operational elements of an object-oriented simulation. It is the interaction of instance objects that simulates the interaction of real-world objects in a system. Instance objects can be created in two ways: sending a class a make instance message, or by using the function make-object.

There are two types of make instance messages: the first creates a named instance belonging to a specified class; the second is similar, but also allows the specification of attribute values. The first message is of the form

(ask *class* make instance *x*)

which results in the creation of a named instance of *class* that is bound to the symbol *x*. As was the case with class names, the symbol *x* is declared to be a global variable and should not be rebound or you may lose your handle on the object. To aid in identification, the print name of the object created is #<*x*>.

The second make instance message is of the form:

(ask *class* make instance *x* with {*attribute value*}*)

where *attribute* is an instance attribute of *class* and *value* is the value assigned to that attribute. For example,

```
(ask fish make instance george with
    genus    betta
    species   splendens
    common-name  siamese-fighting-fish)
```

will return the object **george**, with print name #<GEORGE>. Any attributes that are not specified in the attribute-value portion of the message will take their normal default values. In this example, **george** will have a default length of NIL.

The function *make-object*, of the form

(make-object *class &rest attributes*)

creates an anonymous instance of type *class*. An anonymous instance does not have a name associated with it, is not bound to any symbol, and prints itself as

#<unnamed instance of *class*: *N*>

where *N* is a unique serial number which can be used to visually tell if two anonymous objects are the same. The &rest arguments must be keyword-value pairs that specify the initial value of instance attributes. A keyword is simply an attribute name prefixed by a colon. Thus,

```
(make-object fish
    :genus 'betta
    :species 'splendens
    :common-name 'siamese-fighting-fish)
```

will create an anonymous instance of **fish** with the same attribute values as **george**.

The single argument predicate *instance-object?* returns true if an object is an ERIC instance. Instance objects have three predefined attributes: *parents*, *schedule*, and *status*. Parents is a single element list pointing to the class of which the instance is a member.

11

Chapter 2

Behaviors and Messages

2.0 Message Passing

Recall that objects have behaviors which can perform
computations. Behaviors are pieces of procedural code, each of
which is identified by a message, which is a sequence of symbols.
Behaviors are invoked by a form of indirect function calling
known as message passing. When an object is passed a message,
the behavior identified by that message is executed.

A message is passed to an object with the **ask** function.
Borrowing from the examples in Chapter 1, **george**'s common name
could be determined by sending this message:

(ask george recall your common-name)

to which **george** would reply siamese-fighting-fish. The general
form of the **ask** function is:

(ask object message)

where object is either an ERIC class or instance object, and
message is the sequence of symbols denoting the message being
sent to the object.

2.1 Defining Behaviors

Suppose we would like to provide members of the class **fish** with a set of behaviors which might reflect how a fish actually lives and behaves. A behavior for perpetuating the species could be defined by:

```
(ask fish when receiving (perpetuate the species)
    (ask myself find a fish of opposite sex)
    (ask myself perform courtship behaviors)
    (ask myself settle down and raise kids))
```

There are several important things to notice in this example.

First, we have just defined a behavior for members of the class **fish** that is associated with the message "perpetuate the species". Behaviors are defined by a message of the form:

(ask class when receiving pattern $(action^{*})$)

where pattern is a list specifying the message name and action is zero or more lisp forms that are to be evaluated when the behavior is invoked.

Second, the use of 'myself' in the class position of the messages inside the behavior being defined requires an explanation. The variable **myself** has a special meaning in ERIC. Each time a message is passed, **myself** is bound to the object the message is being sent to. For example, during the evaluation of:

(ask george perpetuate the species)

the variable **myself** will be bound to the instance object george.

13

Myself allows the definition of behaviors that are applicable to any member of a class, without having to know the identity of the member.

Third, when a member of **fish** is sent this message, it will send out three more messages to itself: find a mate, perform courtship, and have children. The three messages sent out by our example must also have behaviors defined to handle them. ERIC behaviors are not limited to sending messages to **myself**. Inside a behavior you can send messages to any object, be it an instance or class.

[There is one other global variable in ERIC that has a special meaning: ***message-sender***, which is bound to the object which sent the current message. If the message was not sent by an ERIC object, ***message-sender*** is bound to the symbol 'user'.]

2.2 Messages are Not Evaluated

Ask is a lisp macro which evaluates only its class argument; the message argument is not evaluated. If you wish to include forms that get evaluated in a message, you must mark the forms with <u>evaluation prefix characters</u>. These special characters tell ERIC that the form directly following them should be evaluated, and the result should be spliced into the message sequence.

14

There are two evaluation prefix characters in ERIC: the exclamation mark (!), and the ampersand (&). Both of these characters specify that the form immediately following them is to be evaluated, but the two characters differ in how they specify the result is to be spliced into the message sequence. The exclamation mark specifies that the result is to be spliced in "as is." The ampersand should only be used with forms that evaluate to a list; this list is then "unwrapped" and spliced into the message. Example 2 illustrates how the evaluation prefix characters work.

2.3 Messages are Actually Patterns

Behaviors which could respond to classes of messages would be more useful than behaviors that can only respond to just one specific message. Take as an example the "ask george recall your common-name" message. Without some way to define behaviors for classes of messages, it would be necessary to write a different

Example 2
Ask Form Evaluation

```
(ask thing go to !place)  ==> (ask george go to tahiti)
     thing = george
     place = tahiti

(ask object go to &place)  ==> (ask george go to the south sea isles)
     object = george
     place  = (the south sea isles)

(ask mammal &(list 'growl) !(+ 1 2) times)
          ==> (ask mammal growl 3 times)
```
15

behavior for recalling every attribute an object has. This could be quite tedious. It is desirable to have a single behavior that would be responsible for handling all messages of the form "recall your *x*".

Writing behaviors that handle classes of messages is accomplished in ERIC by using <u>pattern variables</u> in the message pattern of a behavior definition. The actual definition of the behavior for recalling the value of an object's attribute is:

```
(ask something when receiving (recall your >attribute)
    (object-get myself attribute))
```

Attribute is a pattern variable in this definition.

Pattern variables are denoted by the pattern matching prefix characters > and +. (The greater-than sign and the plus sign, respectively.) When used in a pattern, these prefix characters act as wildcards -- they match against anything. The > prefix will match single forms such as an atom or a parenthesized list. The + prefix will match any number of consecutive forms. If a symbol is prefixed by either of these characters in a pattern, the matcher will bind the symbol to the matched form. Example 3 gives some examples of how pattern matching prefix characters work. In this example, *attribute* is prefixed by the > character, so if a match occurs, *attribute* will be bound to the attribute whose value is being requested. Pattern variables perform the role of formal parameters for behaviors.

16

Pattern	Datum	Results
(a b c)	(a b c)	match, no bindings
(a b c)	(a 1 c)	no match
(a >b c)	(a 1 c)	match, b = 1
(a >b c)	(a (1 2) c)	match, b = (1 2)
(a > c)	(a 1 c)	match, no bindings
(a >b c)	(a 1 2 c)	no match
(a +b c)	(a 1 2 c)	match, b = (1 2)
(a >b c)	(a c)	no match
(a +b c)	(a c)	match, b = ()
(a +)	(a 1 2 c)	match, no bindings
(>a +b c >d)	(e f c g)	match, a = e, b = (f), d = g

<u>Matching Prefix Characters</u>
<u>Example 3</u>

2.4 Inheritance of Behaviors

Objects inherit behaviors from their superclasses, just like they inherit attributes. The mechanism is fairly simple and is based on an object's inheritance chain. When an object is sent a message, it looks to its parent classes for a behavior with a pattern that matches the message. Each superclass on the inheritance chain is checked, from left to right, until a matching behavior is found. If no suitable behavior is found, ERIC signals an error.

Going back to the objects defined in Example 1, let's evaluate the following behavior definitions:

```
(ask something when receiving (move)
    (print 'moving))
```

17

```
(ask fish when receiving (move)
    (print 'swimming))

(ask mammal when receiving (move)
    (print 'walking))
```

After evaluation, **something**, **fish**, and **mammal** will all have a
pattern-action pair for the message "move" in their repertoire of
behaviors. If we now evaluate:

```
(ask george move)
```

the result is that 'swimming' is printed. **George** looked to its
parent, **fish**, for a matching behavior, found it, and evaluated
it. If we ask **fish** or **mammal** to move, the result is that
'moving' is printed. You might have expected **fish** to print
'swimming' and **mammal** to print 'walking', but remember: an object
looks to its parent classes for its behaviors. The parent class
for both **fish** and **mammal** is **something**, so **something**'s move
behavior is evaluated. If we ask **marine-mammal** to move,
'swimming' is printed.

It may seem strange that a class object doesn't look to
itself for a matching behavior, but there is a simple reason why
it doesn't. In ERIC, a class defines a set of objects -- it is
not a member of that set. A class object is a member of its
parent classes, so that is where it should look for its
behaviors. Because all objects search for behaviors by looking
to their parents, both instance objects and class objects have
the same behavior inheritance mechanisms and act in a uniform
manner.

2.5 Before and After Daemons

A <u>daemon</u> is a piece of code that is automatically invoked when some specified event occurs. In ERIC there are two kinds of daemons: <u>before daemons</u>, which are evaluated before a specific message is handled; and <u>after daemons</u>, which are evaluated after a specific message has been handled.

Daemons are useful in a variety of ways; two examples of which will now be given. Often a behavior is defined for a given class, say x. Each of the subclasses of x would like to use this behavior, but a few of the subclasses need to perform different preparatory actions before this behavior is evaluated. Each subclass of x that needs to perform preparatory actions can define a before daemon to do them. Daemons are also useful for uncoupling I/O functions, such as graphics, from behavior defini- tions. Because the I/O is separated from behaviors, it is easy to run simulations with or without graphics and to convert simulations to work with a different I/O device.

Deamons are defined in a way similar to behaviors. Before daemons are defined by a message of the form:

(ask *class* before receiving *pattern* (*action**))

and after daemons are defined by the message:

(ask *class* after receiving *pattern* (*action**))

A class can have only one before and one after daemon for each

message pattern. It is not necessary for a class to have a behavior that handles this message, as long as one of its super-classes does.

The model for ERIC's message handling mechanism developed in Section 2.4 must now be modified to include daemons. When an object is sent a message, the parent classes on the object's inheritance chain are searched from left to right for before daemons that match the message. If a matching before daemon is found, it is evaluated and the search continues with the next class in the chain until all the superclasses on the inheritance chain have been searched. Then a search is made for the behavior which will handle the message (henceforth also called the primary behavior), which is made according to the description given in Section 2.4. After the proper behavior has been evaluated, the parent classes on the object's inheritance chain are searched from right to left for after daemons that match the message. If a matching after daemon is found, it is evaluated and the search continues with the next class in the chain until all the super-classes on the inheritance chain have been searched.

Note that all the before and after daemons defined along an object's inheritance chain for a given behavior are executed, and that after daemons are executed in the reverse order of before daemons. Imagine that the inheritance chain is a path you are walking along. You start at the beginning of the path (the left end of the chain) and walk to the end of the path (the right end

of the chain), stopping along the way to evaluate any appropriate
before daemons you find. While you search for appropriate before
daemons, you also look for the correct behavior to handle the
message. When you come to the end of the path, you evaluate the
behavior you found, and turn around to walk back to the beginning of
the path. As you walk back, you evaluate any appropriate after
daemons you find as you come across them. Once you reach the
beginning of the path again, your journey is finished.

An example of daemon inheritance is in order. Continuing in
the spirit of Example 1, suppose the following forms are
evaluated:

```
(ask fish before receiving (move)
   (print 'before-fish-move))

(ask fish after receiving (move)
   (print 'after-fish-move))

(ask mammal before receiving (move)
   (print 'before-mammal-move))

(ask mammal after receiving (move)
   (print 'after-mammal-move))

(ask marine-mammal before receiving (move)
   (print 'before-marine-mammal-move))

(ask marine-mammal after receving (move)
   (print 'after-marine-mammal-move))
```

If you now send **george** a message to move, the following would be
printed:

```
before-fish-move
swimming
after-fish-move
```

A more complex example would be to send an instance of
marine-mammal a move message, which would result in this being
printed:

```
before-marine-mammal-move
before-fish-move
before-mammal-move
swimming
after-mammal-move
after-fish-move
after-marine-mammal-move
```

Daemons do not return any values. They are useful for side-
effect only. The value returned by a message pass is the value
returned by the behavior which handled the message, regardless of
any daemons that may have been evaluated. Daemons may use para-
meter values that are accessible via pattern variables used in
the *pattern* part of their defining message.

2.6 Wrappers

The final behavioral concept in ERIC that will be discussed
is the wrapper. Daemons let you put code before and after the
execution of a behavior; wrappers allow you to put code around
the execution of a superclass' behavior. Wrappers are not a
distinct class of procedural entities like before and after
daemons are; instead, they are primary behaviors that use the
continue-passing macro to continue searching the inheritance
chain for message handlers.

The *continue-passing* macro can be called either with no
arguments or with one argument. In the no argument case,
continue-passing allows the behavior currently handling a message
to execute the next behavior along the inheritance chain capable
of handling the same message. Consider the following simple
example, based on the objects defined in Example 1. Several new
behaviors are added by sending the following messages:

```
(ask something when receiving (print >n)
    (print 'something)
    (princ n))

(ask fish when receiving (print >n)
    (print 'fish)
    (princ n)
    (continue-passing))

(ask mammal when receiving (print >n)
    (print 'mammal)
    (princ n))
```

After defining these new behaviors, we send **george** a message
to print the number 5:

```
(ask george print 5)
```

and the resulting output is:

```
fish 5
something 5
```

The "print 5" message was first handled by fish's "print >n"
behavior, which printed the first line of output, and then the
call to *continue-passing* sends the "print 5" message on up the
inheritance chain to **something**, which then prints the second line
of output. Similarly, if the same message is sent to marine-
mammal:

23

```
(ask marine-mammal print 5)
```

the resulting output is:

```
fish 5
mammal 5
```

because after **fish**, the next superclass on **marine-mammal's** inheritance chain that has a handler for the "print 5" message is **mammal**.

The message sent up the inheritance chain by a wrapper can be altered by including a new message as an argument to *continue-passing*. If we redefine **fish's** print behavior to be:

```
(ask fish when receiving (print >n)
    (print 'fish)
    (princ n)
    (continue-passing (print !(- n 1))))
```

and send the message:

```
(ask george print 5)
```

the result now is:

```
fish 5
something 4
```

Note that the print message was changed by calling the *continue-passing* macro with the new message to be sent up the inheritance chain as its argument.

Continue-passing is a powerful mechanism for side-stepping normal message handling in ERIC. The new message passed up the inheritance chain can be anything you desire; it is not limited to being a variant of the original message. You should use this

24

mechanism with discretion, however, because its run-time behavior may be difficult to comprehend by just looking at the code. Note that you can only use *continue-passing* at most once in a behavior. It will not work correctly if you try to use it twice.

Chapter 3

Predefined Behaviors

3.0 Behaviors Defined for Something

This chapter describes the predefined behaviors in ERIC.
These behaviors have been defined for the object **something**, so
every new object you define will inherit these behaviors.

3.1 Making Instance Objects

make instance >ob

Creates a named instance object of the class that was sent
the message. The instance is bound to the symbol specified by
ob. See Section 1.2 for more information. Applicable only to
class objects; instance objects will signal an error.

make instance >ob with +attributes

Does everything the previous message does, but allows you to
assign initial values to the newly created instance's attributes.
Attributes is one or more attribute-value pairs. See Section 1.2
for more details and an example. Applicable only to class
objects; instance objects will signal an error.

26

If *ob* already exists as an instance object when either of these messages are sent, the old *ob* is "erased," in the sense that it is removed from its parent's list of offspring and its *schedule* is set to NIL (see Chapter 4 for the significance of this action). However, the old *ob* does still exist somewhere in the lisp environment, so that any pointers you had to it will still be intact. If *ob* already exists as a class object when either of these messages are sent, an error is signalled.

After an instance is created, it is sent an "initialize yourself" message. The default behavior for this message does nothing.

3.2 Defining Behaviors and Daemons

when receiving >pattern +actions
before receiving >pattern +actions
after receiving >pattern +actions

These messages allow you to define primary behaviors, before daemons, and after daemons that are local to the class object receiving them. *Pattern* is the message template that will be associated with the *actions* code. See Section 2.1 for more details and an example. Applicable only to class objects; instance objects will signal an error.

forget your local behaviors matching >pattern
forget your local before daemons matching >pattern
forget your local after daemons matching >pattern

These messages allow you to delete local primary behaviors, before daemons, and after daemons whose invoking message matches *pattern*. These behaviors return an integer telling the number of behaviors that were deleted. Applicable only to class objects; instance objects will signal an error.

3.3 Print Functions

One of the most important ways an object can interact with the user is to display information about its current state. There are several messages for telling an object to print various information.

print yourself

The object will display all of its attributes and values. You get a look at the data structure representing the object. For class objects, this includes the object's behaviors, daemons, immediate descendants, and immediate parents.

print your attributes

The object will display all of the class or instance attributes that were declared in the object's definition. This

28

message is useful when you are only interested in seeing the attributes _you_ gave the object, and don't care about the system defined attributes. The information printed by this message is less cluttered than that printed by the previous message.

print your >attribute

Prints the value of the attribute specified by the parameter *attribute*.

print your local messages

Prints the local behaviors and daemons defined on this object, in alphabetical order. If the object is a class, the behaviors and daemons that have been defined on the class are printed; if the object is an instance, the behaviors and daemons defined on its parent class are printed.

print your messages

Similar to the previous message, but prints every message an object has a handler for. This includes behaviors and daemons that have been defined for an object and all those it inherits from its superclasses, except for the primary behaviors inherited from the superclass **something**. **Something** has so many behaviors that they clutter the screen and obscure the behaviors you have defined.

29

print your messages matching >pattern

Prints all of the object's behaviors and daemons capable of handling messages matching *pattern*. *Pattern* is a list which may contain wildcards as described in Chapter 2. For example, to see all the messages **something** can respond to that begin with the symbol *print*, you would send the message:

(ask something print your messages matching (print +))

3.4 Recalling the Value of Attributes

Printing the values of attributes is nice for human-object interaction, but not very useful for object-object interaction. These recall behaviors return values that can be used in programs.

recall your >attribute

Returns the value of the attribute specified by the *attribute* parameter.

recall the >attribute for your class

Returns the value for a class attribute, specified by the *attribute* parameter, of a class to which the object is a member. If the object sent the message is an instance, the class attribute will belong to its parent class. If the object is a

class object, the attribute value returned will be from the first superclass on the object's parent list that has this attribute. If *attribute* cannot be found for a class, an error is signalled.

recall your local messages

Returns a list of the local behaviors and daemons defined on the object. If the object is a class, the behaviors defined on this class are returned; if the object is an instance, the behaviors defined on its parent class are returned.

recall your messages

Returns a list of all the behaviors and daemons that are available to the object, including those which it has inherited from its superclasses.

recall your messages matching >pattern

Returns a list of all the behaviors and daemons available to the object that handle messages matching *pattern*.

recall your descendants

Returns a list of all the descendants of a class. Descendants are defined to be instances and all subclasses and their instances. Applicable only to class objects; instance objects will signal an error.

31

recall your instances

Returns a list of all the instances belonging to this class or one of its subclasses. Applicable only to class objects; instance objects will signal an error.

recall your subclasses

Returns a list of all the subclasses of the object. Applicable only to class objects; instance objects will signal an error.

recall your superclasses

Returns a list of all the superclasses of an object.

3.5 Setting Attribute Values

set your >attribute to >value

This behavior is for assigning a value to an attribute. This works for both instance attributes and class attributes. If the object does not have *attribute* as one of its defined attributes, an error is signalled.

set the >attribute for your class to >value

Assigns *value* to the class attribute specified by
attribute. Applicable only to instance objects; if this message
is sent to a class object, an error will be signalled. If the
object's class does not have *attribute* as one of its defined
class attributes, an error is signalled.

3.6 Sending Messages to Attributes

to ask each of your >attribute to +action

Sends the message *action* to each element in the value of the
attribute specified by *attribute*. The attribute's value should
be a list. For example, to have each object in the offspring
attribute of **something** print itself, you would send this message:

(ask something to ask each of your offspring to pr_nt your self)

to ask your >attribute to +action

Sends the message *action* to the object that is the value of
the attribute specified by *attribute*.

33

3.7 Tracing and Recording Messages

ERIC provides two ways to observe the passing of messages: tracing and recording. Each of these facilities serves a different purpose. The trace facility is useful for debugging programs and is similar to lisp tracing facilities; the recorder saves a record of messages passed for later examination.

trace your messages matching >pattern

This behavior marks all of the class object's primary behaviors which match *pattern* for tracing. *Pattern* may contain wildcard matching characters. When any object that is a member of this class receives messages which match *pattern*, the trace facility prints four pieces of information to the current standard output stream: the current trace depth, the class that is handling the message, the message itself, and the value returned by the invoked behavior. The trace facility indents nested message passes. This behavior is applicable only to class objects; if sent to an instance object, an error will be signalled.

untrace your messages matching >pattern

This behavior unmarks all of the class object's primary behaviors whose invoking message matches *pattern* so they will not

be traced. This behavior is applicable only to class objects; if sent to an instance object, an error will be signalled.

record your messages matching >pattern to >stream
unrecord your messages matching >pattern

The recording facility is similar to the trace facility, except that there is no indention and only the message passes are recorded, not their results. The current simulation time and the object who sent the message are also included with each message record. *Stream* must already be open for output, and you must close it when you are finished recording.

Chapter 4

Event Scheduling

4.0 The CLOCK

The features of ERIC described so far have only dealt with
its object-oriented programming capabilities. Objects alone do
not a simulation make. There must also be a temporal control
mechanism which allows objects to interact over time. In ERIC
this mechanism is the **clock** object. The **clock** controls the flow
of time in the simulation and also allows actions to be scheduled
for execution in the future. The **clock** is an instance object of
class **simulation-clock**, and has three instance attributes:
simtime, *event-list*, and *ticksize*.

Simtime is the current time in the simulation. In ERIC time
is represented as a real number which has no built-in scale of
measure. It is left to the programmer to decide what scale of
measure (if any) is associated with time, and to use this scale
consistently within all object behaviors. For example, if you
wish time to be measured in seconds, you should write behaviors
that consistently refer to time in terms of seconds. Because time
is represented as a real number, there is no indivisible interval
of time in ERIC.

36

Simulations generally keep actions scheduled to happen in the future in some sort of time-ordered queue. In ERIC, this queue is maintained as the **clock**'s *event-list*. The **clock** moves the simulation forward in time by executing events in the queue. The progression of simulation time is not smooth and continuous -- it jumps from one event to the next. The interval of time between events never really exists.

It is often convenient to stop the execution of a simulation at regular intervals, or to run a simulation for a specified period of time. The **clock** will start the execution of a simulation when it receives a tick message:

(ask clock tick)

The simulation will then run for a length of time specified by the **clock**'s *ticksize* attribute, which controls how many time units pass during each tick of the **clock**. Changing the value of *ticksize* is one way you can control how long a simulation runs. Another way is to use the message:

(ask clock tick >n times)

which is simply a loop which sends **clock** a tick message *n* times.

The "ideal" value of *ticksize* varies from simulation to simulation. Bear in mind that the value of *ticksize* has no direct effect on the behavior of the simulation; the **clock**'s ticking is only for interaction and control purposes. Generally, the tick size should be longer than the mean time between events

37

in a simulation. There is not much purpose in stopping every
second of simulation time if the mean time between events is
several minutes of simulation time.

4.1 Scheduling Events

There are two basic messages for scheduling actions to
happen at some future time:

(ask clock to schedule >object to +action at time >x)

(ask clock to schedule >object to +action in >x time units)

The first message schedules an action to happen at some absolute
time; the second message schedules an action to happen relative
to the time at which the message was sent. *Action* must be a
message that *object* can handle, or there will be an error
signalled at some point in the future when the object tries to
execute the action.

A small example of scheduling is in order. Suppose we want
a behavior that makes a member of the **fish** class move every 10
seconds. Notice that time was referred to in terms of seconds;
you can assign any scale of measure to simulated time by simply
deciding to use that scale consistently throughout your simula-
tion code. To make the use of seconds explicit, a new scheduling
behavior for **clock** is written:

```
(ask clock when receiving (to schedule >object
                                to +action in >x seconds)
    (ask clock to schedule !object to &action in !x time units))
```

In a similar way, you can define behaviors for dealing with time in whatever scale of measure you like. Now we can write the new behavior for **fish**, which will be invoked by the message "move around":

```
(ask fish when receiving (move around)
    (ask myself move)
    (ask clock to schedule !myself to move around in 10 seconds))
```

This behavior first tells the object send the message to move, and then schedules the object to move around again in ten seconds. We could set **george**, the fish created in Chapter 1, in motion by sending it the message "move around" and then asking **clock** to tick a few times. Every ten seconds **george** would move and then reschedule himself to move around in another ten seconds.

4.2 The Queueing Mechanism in More Detail

There is more to the event queueing mechanism than was described in Section 4.0. A thorough grasp of this mechanism is central to understanding how ERIC works. ERIC's scheduler is a bit unusual in that the event queue is distributed between the clock and the other objects in the simulation.

Studies have shown that simulations spend a large percentage of their run time managing their event queue. Therefore, it is essential that a simulation language provides an efficient queueing mechanism. Managing the event queue mostly consists of two operations: inserting new events into the proper place in the ordered queue, and retrieving the event which should be executed next. Since the queue is time-ordered, the retrieval operation is trivial. Insertion is much more expensive than deletion, with the cost of inserting an event depending on how large the queue is. It is also expensive to delete an event from the queue. However, in most simulations this is a rare operation, and some simulation languages do not support event deletion. Like insertion, the cost of a deletion depends on the event queue's size.

ERIC was designed to support simulations which are composed of intelligent objects. One of the things intelligent objects in the real world do a lot of is build, execute, and modify plans; it is reasonable to assume that intelligent objects in a simulation would need to do the same. Therefore, ERIC should make it easy and efficient for objects to examine and modify their scheduled future events. That means providing an efficient way to search for, insert, and delete events in the event queue. This is where ERIC's distributed queueing mechanism comes in.

As mentioned earlier, the length of the event queue is an important factor in how long it takes to perform search, insert, and delete operations. Therefore, it is desirable to keep the

event queue as small as possible. The first part of ERIC's queue mechanism, the clock's *event-list* attribute, keeps track of which objects have actions scheduled and when the actions are to be executed. Actually, the *event-list* only keeps pointers to a small number of the (possibly many) events scheduled for a given object. Ideally the *event-list* will contain only one reference to a given object, but this is not always possible. Keeping only a few pointers to each object helps reduce the size of the *event-list*.

Each object's *schedule* attribute contains information about its scheduled future actions. *Schedule* stores a time ordered list of data structures called <u>plans</u>, which contain three pieces of information: what action is to be performed, when the action is to be performed, and who scheduled the action. Allowing each object to store its own plans facilitates the examination and modification of plans. It is far more efficient to have an object rummage through its *schedule* attribute, which contains only its own scheduled future actions, than to have it rummage through a monolithic event queue that contains every object's future actions.

When an action is scheduled for the object X, the clock checks X's *schedule* for plans. If X currently has no plans in its schedule, or if the new action is supposed to happen before any event currently in the schedule, the clock places a reference to X's future action in *event-list*. If X currently has actions

41

scheduled to happen before the new action, then there already is
an appropriate reference to X in the *event-list*, so a new
reference is not inserted. (This is how the event-list size is
kept down to about one reference per object.) In either case,
the future action is inserted into the object's schedule.

When the **clock** pulls an event off the *event-list* for
execution, it checks the *schedule* of the object specified in the
event to see if the object has a plan that should be executed at
the current time. If so, it is executed; if not, nothing
happens. In either case, the **clock** then puts a reference to the
object's next scheduled action into the *event-list*. The cost of
checking an object's schedule when there is nothing to do is less
expensive than trying to delete an action from the event queue.
This allows plans to be deleted or modified efficiently.

4.3 Objects Can Die

Another benefit of the distributed event queue is that it is
easy to "kill" an object. In the physical world, objects can be
destroyed and are no longer able to perform actions. All
instance objects in ERIC have a *status* attribute which is either
the symbol 'alive' or 'dead'. An instance object can be killed
by sending it the message:

(ask *object* kill yourself)

42

Unlike most dead objects in the physical world, dead ERIC objects still exist and are able to respond to messages sent to them. However, they cannot be scheduled to perform any actions in the future and they will not perform any actions that had been scheduled prior to their death. If you try to schedule an action for a dead object, an error will be signalled. It may be necessary in your application to perform certain actions when an object dies; remember that you can use daemons or wrappers for this.

Two lisp predicate functions, *live-object?* and *dead-object?*, tell you if an object is alive or dead, respectively. You should avoid modifying the *status* attribute in your own code. If you need to keep information about an object's "status" with respect to your application, use an instance attribute of your own making.

4.4 Plan Manipulation

As mentioned earlier, the future scheduled actions for an object are stored in the object's *schedule* attribute in the form of plans. Each plan is a data structure which holds three pieces of information: the action to be performed, the time when the action should be performed, and who scheduled the action. These pieces of information can be retrieved from a plan with the

43

functions *get-plan-action*, *get-plan-time*, and *get-plan-scheduler*,
respectively. Each of these functions takes a plan as their sole
argument.

Several behaviors are provided that allow you to manipulate
an object's *schedule*. They are:

forget your plans matching >pattern

Removes all of an object's plans whose action matches
pattern from *schedule*. The matching plans will not get executed.
Pattern may include any number of wildcard pattern variables.

forget your plans at time >x

Forgets all of an object's plans that are scheduled to
happen at time *x*. These plans will not get executed.

forget your plans before time >x

Forgets all of an object's plans that are scheduled to
happen before (but not including) time *x*. These plans will
not get executed.

forget your plans after time >x

Forgets all of an object's plans that are scheduled to
happen after (but not including) time *x*. These plans will
not get executed.

44

forget your plans between times >time1 and >time2

Forgets all of an object's plans that are scheduled to happen between times *time1* and *time2*, inclusive. These plans will not get executed.

If these behaviors do not satisfy the needs of your application, you can write your own. You are allowed only to remove plans from an object's *schedule*; if you wish to add them, you must send scheduling messages to the clock. *Schedule* contains a list of plans, ordered from the earliest to latest time of scheduled execution. Remember to maintain this ordering when you modify *schedule*, or things will go awry.

Chapter 5

Miscellaneous Matters

5.0 Purpose

The purpose of this chapter is to present several important short topics that do not fall neatly into the previous chapters. Most of these topics deal with lisp functions that are useful for programming in ERIC.

5.1 Objects are Structures

ERIC objects are implemented as CommonLisp structures. Any of the facilities provided by CommonLisp for dealing with objects may be used with ERIC objects. The typing system is of particular importance. All class objects in ERIC are structures of the type 'class-object'. Instance objects are also structures, but they may come in a variety of types.

When a class is created via define-object, ERIC builds a CommonLisp defstruct form appropriate for implementing instance objects of that class. One of the many things this defstruct form does is create a new lisp data type with the same name as

the ERIC class name. This name can be used with the CommonLisp function *typep* and *type-of*, so that

     (typep george 'fish)

is true and

     (type-of george)

returns 'fish'.

The defstruct also defines accessor functions for each of an object's instance attributes. (See the CommonLisp language definition for more information on defstruct.) Accessor functions perform the same job as the "recall your >attribute" and "set your >attribute to >value" messages, but accessor functions are much more efficient. However, accessor functions are suitable only in special situations and should be used with extreme care. The reason for this lies in CommonLisp's limited inheritance mechanism for structures.

CommonLisp's structure facility is not designed to support multiple inheritance of attributes. Therefore, ERIC has to map a class' attributes (some of which may be inherited from multiple superclasses) onto structures. This mapping is not used by the accessor funtions generated automatically by a defstruct, so the accessors don't always behave correctly. The only situation where it is safe to use defstruct-generated accessor function for a given instance attribute is when the class the attribute belongs to will never be combined with another class that has an instance attribute of the same name.

Message passing is a lot more expensive than normal function calling. Therefore, the two messages for accessing object attributes can exact quite a toll on a simulation's run-time performance. Never fear, for ERIC provides two lisp functions that will correctly access any object's attributes (be the object a class or an instance) more efficiently than the messages can. The functions are *object-get* and *object-put*, and they are of the form:

(object-get *object attribute*)

(object-put *object attribute value*)

Using these functions to access attributes will enhance run-time performance, but you lose the flexibility of message passing. You will not be able to take advantage of the daemon, tracing, or recording facilities ERIC provides. You should consider all these factors when deciding which accessing method to use.

5.2 Some Useful Predicates

This section presents some useful lisp predicate functions. They have been presented separately elsewhere in this paper, but are gathered here for convenience. All of the predicates take a single argument.

object? Returns true if the argument is an ERIC object.

48

class-object? Returns true if the argument is an ERIC class object, nil otherwise.

instance-object? Returns true if the argument is an ERIC instance object, nil otherwise.

live-object? Returns true if the argument is a live ERIC object, nil otherwise.

dead-object? Returns true if the argument is a dead ERIC object, nil otherwise.

5.3 Matching Facility

The pattern matching facility of ERIC is available for you to use via the *ematch* function, of the form:

(ematch *pattern datum*)

which matches the *datum* against the *pattern*. *Pattern* may contain pattern variables and wildcards. There are two caveats when pattern matching: the same pattern variable should not appear twice in the same pattern; and pattern variables should not appear immediately after a + or +var wildcard.

The result returned by *ematch* is one of three possibilities: NIL, if there is no match; T, if the match is successful but there are no variables in the *pattern*; or an association list of variables and bindings, if the match is successful and there are variables in *pattern*.

49

FURTHER READING

If you are interested in reading more about some of the topics
covered in this report, these references may be of interest.

ROSS: An Object-Oriented Language for Constructing Simulations,
Rand Note R-3160-AF, The Rand Corporation, 1984.

The ROSS Language Manual, Rand Note N-1854-AF, The Rand
Corporation, 1982.

Object-Oriented Programming: Themes and Variations, Mark Stefik
and Daniel G. Bobrow, The AI Magazine, Winter 1986, pp. 40-62.

Common Lisp: The Language, Guy L. Steele Jr., Digital Press,
1984.
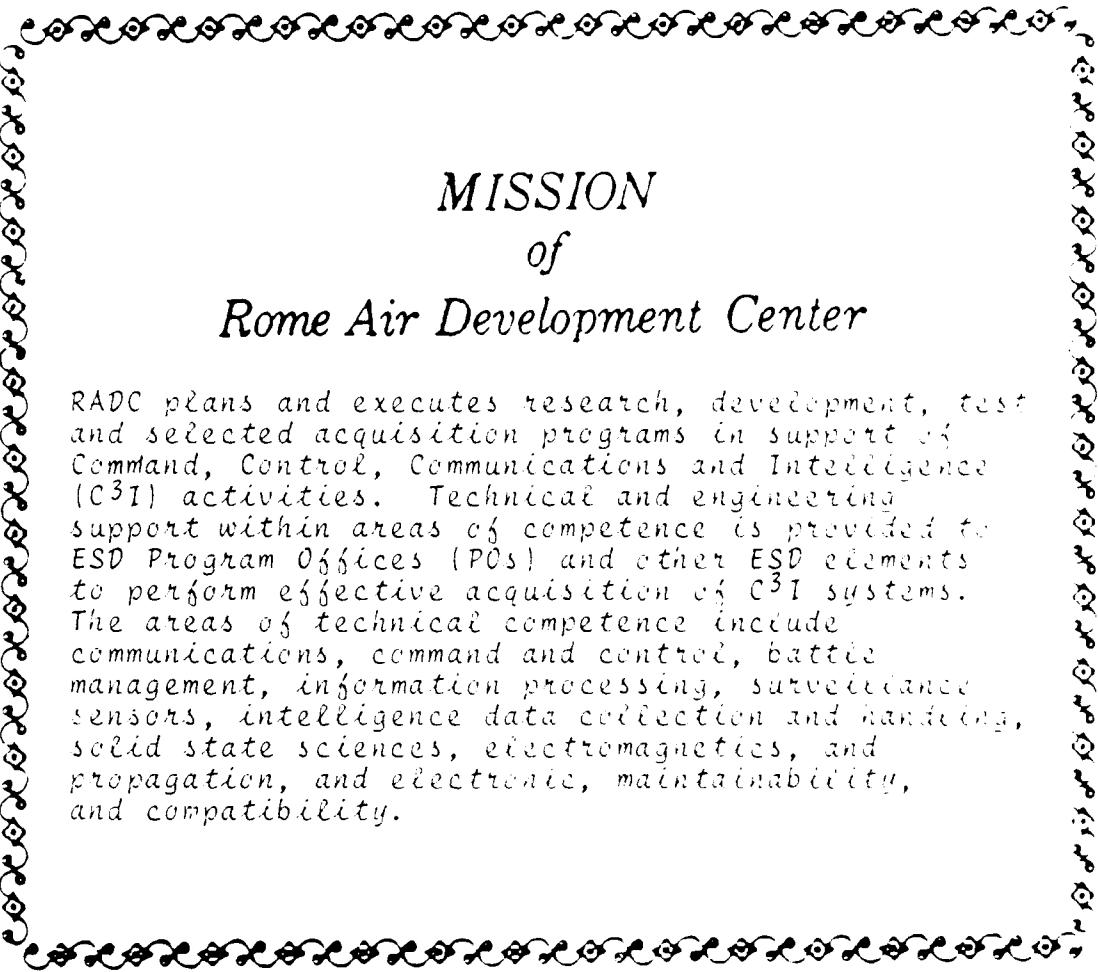
MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.

END

10-87

DTIC